(2)

RADC-TR-90-404, Vol XVI (of 18)
Final Technical Report
December 1990

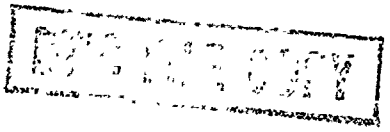
INTELLIGENT SIGNAL PROCESSING TECHNIQUES FOR MULTI-SENSOR SURVEILLANCE SYSTEMS

Northeast Artificial Intelligence Consortium (NAIC)

Harvey E. Rhody and Robert T. Gayvert

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

This effort was funded partially by the Laboratory Director's fund.



Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-404, Vol XVI (of 18) has been reviewed and is approved for publication.

APPROVED:

VINCENT VANNICOLA
Project Engineer

APPROVED:

JAMES W. YOUNGBERG, Lt Col, USAF
Deputy Director of Surveillance

FOR THE COMMANDER:

BILLY G. OAKS
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (OCTS) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1990	Final Sep 84 - Dec 89

4. TITLE AND SUBTITLE
INTELLIGENT SIGNAL PROCESSING TECHNIQUES FOR MULTI-SENSOR SURVEILLANCE SYSTEMS

5. FUNDING NUMBERS
C - F30602-85-C-0008
PE - 62702F
PR - 5581
TA - 27
WU - 13
(See reverse)

6. AUTHOR(S)
Harvey E. Rhody and Robert T. Gayvert

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Northeast Artificial Intelligence Consortium (NAIC)
Science & Technology Center, Rm 2-296
111 College Place, Syracuse University
Syracuse NY 13244-4100

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COES)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-404, Vol XVI (of 18)

11. SUPPLEMENTARY NOTES (See reverse)
RADC Project Engineer: Vincent Vannicola/OCTS/(315) 330-4437
This effort was funded partially by the Laboratory Director's fund.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Northeast Artificial Intelligence Consortium (NAIC) was created by the Air Force Systems Command, Rome Air Development Center, and the Office of Scientific Research. Its purpose was to conduct pertinent research in artificial intelligence and to perform activities ancillary to this research. This report describes progress during the existence of the NAIC on the technical research tasks undertaken at the member universities. The topics covered in general are: versatile expert system for equipment maintenance, distributed AI for communications system control, automatic photointerpretation, time-oriented problem solving, speech understanding systems, knowledge base maintenance, hardware architectures for very large systems, knowledge-based reasoning and planning, and a knowledge acquisition, assistance, and explanation system.

The specific topic for this volume is intelligent signal processing techniques for multi-sensor surveillance systems.

14. SUBJECT TERMS
Artificial Intelligence, Signal Processing, Multi-Sensor Surveillance Systems, Simulation

15. NUMBER OF PAGES
60

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

NSN 7540-01-280-5500

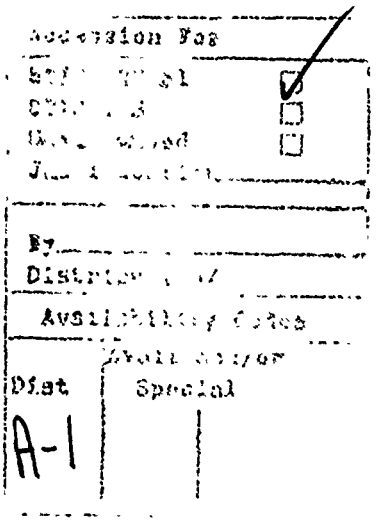
Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Block 5 (Cont'd) Funding Numbers

PE - 62702F	PE - 61102F	PE - 61102F	PE - 33126F	PE - 61101F
PR - 5581	PR - 2304	PR - 2304	PR - 2155	PR - LDFP
TA - 27	TA - J5	TA - J5	TA - 02	TA - 27
WU - 23	WU - 01	WU - 15	WU - 10	WU - 01

Block 11 (Cont'd)

This effort was performed as a subcontract by the RIT Research Corporation to Syracuse University, Office of Sponsored Programs.



Contents

Chapter 1

Introduction

This is the final report on a study of intelligent signal processing techniques for multi-sensor surveillance systems. The project was carried out under the RADC Northeast Artificial Intelligence Consortium under contract F30602-85-C-0008. The study was done during the period May 1989 through December 1989.

1.1 Participants

The work reported in this document was done at the Rochester Institute of Technology and coordinated with related work at the University of Buffalo and the Rensselaer Polytechnic Institute. A previous study under the RADC Expert Scientist and Engineering Program included all three institutions within the same effort.

1.2 Background

Current surveillance systems must perform a variety of tasks within a complex real-time environment. Artificial intelligence (AI) techniques which have been developed for modern signal processing applications such as vision, image understanding and speech understanding combined with other AI techniques such as expert systems, knowledge representation, plan recognition, search and control offer ways to improve the performance of the next generation of surveillance systems.

Current surveillance systems make use of extensive signal processing for target detection, tracking and recognition. The signal processing has been optimized for the given target/sensor combinations. It is likely that the next generation of systems will make use of these sensors and their processing algorithms but that the information will be used in a different way. The information from a number of sensors will be combined by a higher-level system to provide enhanced detection, tracking and recognition as well as the ability to recognize threats and characteristic or uncharacteristic behavior in complexes of targets.

The next generation of systems are likely to make use of a distributed set of sensors and processors. This approach offers the greatest modularity and flexibility in system development, deployment and maintenance. If effective custom systems can be constructed from a set of generic modules then there will be a very substantial savings over the cost of individual custom systems for a variety of applications.

Intelligent systems offer systematic processes to address problems such as sensor fusion, sensor coordination, threat assessment, decision analysis and resource allocation. All such tasks require that information be handled at a high level so that symbolic reasoning can be supported.

The behavior of distributed systems with significant numbers of interacting components is difficult to analyze and predict. Even if the individual system elements are well-understood, a system composed of many of them may exhibit new and unexpected modes. The inclusion of nonlinear processes, as decision processes must be, makes the theoretical analysis of such systems essentially impractical. Therefore the behavior and performance of distributed systems can be best evaluated by the use of prototypes and simulations.

The purpose of this project is to extend the framework of the modern multi-target, multi-sensor surveillance environment and to investigate the adaptation of intelligent signal processing algorithms to that application.

A system to simulate the radar environment, including transmitters, sensors, targets, noise and stationary objects has been designed and a prototype has been constructed. This system has been constructed as a discrete event simulator, which permits maximum flexibility and efficiency in run-time computation. It has the potential to be extended to concurrent networked operation.

The prototype implementation is called ESSPRIT, and is written in Lisp

using Flavors for TI Explorer computers. The implementation is described in Appendix A, and the technical details are contained in Appendix B. The system uses a window-oriented environment to construct and operate simulations. It is a general-purpose system that will permit simulations to be constructed for many kinds of applications.

A simulation system consists of the basic environment which allows objects to be created, to interact over time, to be observed and reports to be created. These elements have been created and a basic set of radar simulation objects (signals, sensors, targets) have been implemented and tested. The details of the implementation, construction of a simulation, and operation are contained in the appendices.

Chapter 2

Discrete Object-Oriented Simulation

Discrete event simulation (DES) is a computerized technique for experimenting with models of physical systems. It allows one to investigate the impact that changes in individual systems elements have on the performance of the total system. If the simulation system is sufficiently flexible and powerful, it may also promote investigations of different system structures. Both kinds of investigation will be promoted by a simulator tool which makes it easy to model and interconnect system elements.

As an example, consider an investigation of the ability of an automatic pilot to control an airplane in a terrain avoidance application. A basic simulator would require a model of the terrain, the dynamic behavior of the airplane, the senors and the control system. The behavior of the total system would be determined by the interaction of these modules. An improved control system could be tested by substituting its model for the existing control system model in the simulator. So long as the system protocols were met, the simulator should continue to function properly and thereby provide insight about the performance improvement gained by the new controller. One could test the performance of the controller with different aircraft or different terrain by changing those modules and observing the new behavior. Object-oriented tools make it possible to assure that the simulator protocols are met.

In some cases performance can be improved more effectively by changing the system structure than by improving individual elements. As an exam-

ple, it may be possible to improve the detection and tracking capability of a surveillance system either by improving the performance of the existing sensors or by changing the structure of the system so input from other kinds of sensors can be used. Some information that is gathered in another form, perhaps in another time or location, may be available at a much lower price than the cost of improving and retrofitting the existing sensors. The diversity of information sources may provide more useful information and may offer better system performance. However, with a new system would come some new design issues. In particular, it would be necessary to provide ways to integrate the new information into the system control structure. It is therefore necessary to understand the information fusion task and the system restructuring that will achieve that fusion. It would be very useful to be able to add modules to the system and investigate various information fusion and control structures. Object-oriented modeling and simulation offers that capability.

Simulation permits one to investigate the impact of changes in the structure or parameters of a system without the expense or danger of constructing or modifying a physical manifestation of such a system. However, it would be possible to substitute physical elements for some of the modules of the simulation if the proper physical support and interfacing were provided. This would be made easier by using object-oriented modeling and simulation, as discussed below.

Through the use of simulation one may learn about reliability, throughput rates, bottle-necks, response times, and other aspects of system behavior. Simulation can help designers to decide where to reduce or reallocate scarce resources while maintaining or even improving overall system performance.

A general goal for a simulator is to provide an environment and the tools for building and evaluating prototypes of large heterogeneous software and hardware systems. This capability can be used either to evaluate–and understand–existing systems or to design new systems. The designer should be able to quickly build a prototype of a system and evaluate its performance. The tool will then be a tool for creating designs by evolving prototypes. The prototype evolves through successive stages of refinement as the designer gains insight. At any particular stage, the current prototype is an embodiment of the current design. Design by prototyping is a sound, well-understood and cost-effective method. It is being used in modern software engineering tasks to increase both productivity and quality.

2.1 Object-Oriented Programming

Considered by many to be one of the most significant advances in computer science in many years, Object-Oriented Programming (OOP) provides a methodology and the associated programming language support for programming "in the large." The methodology has existed for several years and is the one recommended for Ada system development. Several existing languages support OOP to one degree or another: the oldest language to be considered an OOP language is Simula67, an Algol derivative meant primarily for simulation; newer languages include Ada, C++, Lisp-Flavors and Smalltalk. Other experimental or research languages exist specifically to study OOP include Act1 and Act2.

In an OOP system, the components are considered to be independent objects that interact by sending and receiving messages. An object is an integrated unit of data and procedures, which are called methods, that act on the data. The object is described by state variables, called instance variables, and the data values are used to specify the state by giving values to the state variables. Because objects can interact only by sending messages, the data is encapsulated and protected. A message can contain any kind of information, including data and methods, and may be sent to any number of objects. Messages themselves are objects. Thus, objects may create other objects.

Upon receiving a message an object may process it and take a number of actions. These include modifying its state–which may include "dying," sending one or more messages to other objects and creating new objects. The new objects are created as instances of types of objects that have been described to the system by the programmer. To create an object a message is sent to the system with the specifications for the object and its state.

Objects are grouped into classes which describe the behavior of a kind of object, an instance of the class. This description includes the nature of the internal data and the methods which can be executed. Subclasses may inherit the structure and methods of a class. Object-oriented classes are polymorphic; i.e. the same message can be sent to both a class and its subclasses. For example, if there was a polyhedra class and cube, prism and tetrahedron were subclasses, all the classes could receive a standard message 'print volume' and respond correctly according to the appropriate method for their geometry.

Objects are grouped together in hierarchical classes which can then be put

into separate modules. Parts of the program outside a given class or module can only interact in specified ways with the class or module and thus does not need to know how given methods are executed, variables changed or classes structured. More primitive ideas can be encapsulated in the super classes and thus reduce the level of complexity visible in the subclasses. Information about a particular structure or implementation can be hidden within an object or class.

Programs in object-oriented languages can be readable and comprehensible. The capacity for multi-level reading of simulation programs, focusing on different levels of the class hierarchy or modules, can give new users a quick overview and more experienced users a in-depth look.

The **ESSPRIT** system (Appendices A and B) provides program visualization in the form of a system block diagram. The blocks are icons which represent simulation objects on the computer screen. Objects can be added to a prototype by selecting them from a menu, placing them on the screen, specifying their initial parameters and connecting them with other objects. The connections define the paths for messages. Groups of objects can be merged into new icons which represent the complex, so that a hierarchical representation of the system can be built. Various dials, gauges, reports and graphs can be selected to provide reports of activity of any objects in the system.

Such a modular structure can ease modification since one module can be altered without affecting others. This can lead to more flexible and extensible programs. It encourages software reuse since modules from different simulation programs can be pulled in when constructing a new program. A user can easily make changes "on the fly;" new methods can be defined or new classes and objects added. Subroutines and utilities from other languages can also be utilized. This modularity gives the user the ability to pull desirable features into an object-oriented simulation program. Large programs can be broken into many small, independently functioning units.

The hierarchical structure allows the prototype to be developed from top-down. High-level objects, representing major subsystems, can be constructed so that the top-level performance can be evaluated. Once the characteristics of the major subsystems have been stabilized, each of them can be prototyped in terms of its internal building blocks. Particular subsystems can be modeled to lower levels than others an action which may be useful where some subsystems are taken to be fairly standard while others are more novel.

Once the prototype is complete it represents a design for an actual physical system.

The use of messages and objects seems to be a natural model for many systems. This style of programming parallels the way one intuitively thinks of processes in dynamic systems. Behaviors are attached to specific objects just as the real world entities exhibit different behaviors. Such a software design can also correlate programming objects on a one-to-one basis with real-world objects. It specifies in one place all the data associated with an object and the routines or methods which can manipulate that data. Such structure can allow both naive and experienced users to quickly understand a model.

The object-oriented design is well-suited for concurrent computing on distributed computers. Objects or modules can be placed on different processors and communicate via message passing. The computing environment can contain different kinds of computers and the actual application code on each computer can be in different languages. The only absolute requirement is that the computers respond to messages in the appropriate way, so that there must be a communication and message-handling interface. Existing simulation code or existing data structures can be wrapped in appropriate message-handling shells and used in a simulation environment. This makes it possible, in some cases, to even use existing machine code–a useful option when the source code no longer exists.

The distributed environment can be extended to non-traditional "computers." Physical hardware can be used for particular components in the simulation by providing the hardware and software interfaces to the computer. This makes it possible to use special processors or existing components of systems to speed-up a simulation or to make it work with some real components. The major consideration in using physical devices is their ability to respond appropriately in simulator time rather than in real time.

A special non-traditional "computer" that can be brought into an object-oriented simulation is a human participant. Messages can be sent to the human through the screen and by sound and replies can be sent to the system by the keyboard and mouse. As far as the system is concerned, the human is just another object in the process. This makes it convenient to communicate with the operator, but also allows people to be actual participants in the simulation. The traditional Macintosh display using icons and windows is one special form of object-oriented programming that facilitates communications

between the operator and the computer.

The interactive nature of many object-oriented languages facilitates intelligent exploration. The program can be interrupted while it is running, the state of the system can be modified–even by adding or deleting components–and the simulation resumed. The state of the system at the time of interruption can be stored so that it can be restarted from that point to explore a variety of options.

Typical systems that are modeled and simulated will consist of several directly and indirectly interacting components. In the real system–the one being modeled–these components are independent; in the simulation the components should be concurrent or operate in parallel if the computer technology supports it. The control system of the DES model allows each model component to proceed whenever its input is ready. One concern that shows up in the simulation environment but not in the real world is that of time management; a component needs to treat as part of its input the fact that the simulated time is correct in order for that component to proceed.

One of the most important problems with traditional computer programs is their inflexibility. Programs need to be changed–this is their most universal aspect. The older languages and methodologies simply enforce this rigidity, often building highly restrictive descriptions int the functional system requirements. Object-oriented programming directly counteracts that aspect of systems, encouraging early prototyping, with a concomitant lack of details. A broad-brush structure or skeleton is constructed easily and details are added later. When the details change, as they are sure to do, they can be easily modified. In contrast, in traditional programming it is the decisions that are made the earliest that are hardest to change since their affects ripple through the rest of the system.

Since object-oriented programming addresses the problems of structuring a large system– programming in the large–it is a particularly good candidate for scaling up to simulate very large systems.

2.2 Discrete Event Simulation

Discrete Event Simulation is particularly valuable in the design of both hardware and software systems. As noted above, Discrete event simulation (DES) is a computerized technique for experimenting with models of physical sys-

tems. It allows one to investigate the impact that changes in individual systems elements have on the performance of the total system. If the simulation system is sufficiently flexible and powerful, it may also promote investigations of different system structures. Both kinds of investigation will be promoted by a simulator tool which makes it easy to model and interconnect system elements.

The ESSPRIT tool discussed in Appendices A and B is intended to provide this kind of environment. It is a simulation shell that permits many different kinds of systems to be simulated. As a part of this project it is being modified to provide a test-bed for multi-sensor radar systems for a multi-target environment.

The multi-sensor multi-target radar environment consists of many directly and indirectly interacting components, as described in Chapter 3. In the real system being modeled these components act independently; in the simulation, then, the components should be concurrent (or operate in parallel if the computer technology supports it). The control system (operating system or run-time system) that directs the overall operation of a DES model permits a component to "run" whenever its input is ready and the simulated time is correct. The concern of time management does not show up in the real world but must be dealt with in the simulated world. A component must treat time as a part of its input so that it may not proceed incorrectly. No component can be allowed to modify the past of any other component.

Object-oriented programming naturally lends itself to Discrete Event Simulation. The objects are the components of the system being simulated. The OOP notion must be modified but slightly: objects send *time stamped* messages to other objects, and the messages must be processed in time-stamped order so that an object can be prevented from modifying the past of another object.

2.2.1 Time Management

In DES there is generally a global clock that gives the simulated time of the system. The primary control is to iteratively wait until the system is quiet ("quiescent"), determine the least time value for the components that are prepared to work, advance the global clock to that time, and start the corresponding process. This is a contrast with continuous simulation in which the clock is stepped by uniform amounts Δt and the simulator integrates

systems of differential equations equations.

In simulations such as a radar environment, different components may operate on widely-varying time scales, from microseconds to hours or days. This wide range makes it difficult to run such simulations with continuous simulators since all equations must be stepped by the smallest time step needed by any component. In DES it is possible to allow only those elements which require processing at each time to proceed, so that there can be a gain in efficiency by orders of magnitude.

Continuous simulation systems put the dependencies between components in the model and DES puts them in the component descriptions by describing component behaviors. In that way DES is much more flexible and modular.

A typical operation in DES may go as follows:

1. A client seizes a resource
2. The client uses the resource for N seconds
3. The client releases the resource

The second step causes the particular process to be "put to sleep" for N simulated time tics; i.e., a wakeup is scheduled for $time = currenttime + N$. (The choice of N often depends upon a pseudo-random number generator to express the analyst's choice of a statistical distribution.)

The wakeup manager plays a role similar to an operating system scheduler or a dataflow system monitor (or imagine a hotel's wake-up service). The principal data structure, which operates like a priority queue, is called the "(future) events queue." This events queue is a set of pairs, $(time, event)$; elements can be added to this set without any particular discipline (except that the *time* component can not be less than the current simulated time), but every time a pair is removed, it is the pair with the lowest *time*. One may picture this as either a linked list or as a sorted array.

When a system component "goes to sleep," one imagines the ordered pair, above, "waiting" as a surrogate for the component in the wakeup manager's queue. (In a similar arrangement a surrogate for a process will wait in a queue—waiting, say, to use a particular resource.)

2.2.2 Miscellaneous Issues for DES

2.2.2.1 Queues

Whenever a client attempts to seize a resource that resource may be in use. The client may be able to test for that possibility, but often there are no alternative actions possible (or planned), so the client must "wait" in a queue. Such queues can be either explicit, with names and having statistics gathered and reported, or implicit components of the resources, simply managed by the system but otherwise invisible. In the latter case, the clients may keep track by themselves of the amount of valuable work accomplished over a period of time an the time lost waiting in queues. Generally queues are FIFO, but other structures are possible as well:

1. Priorities of the clients may affect waiting orders.
2. Queues may have a limited capacity and be unable to accept entries beyond a specified limit.
3. Clients may be able to exit from a queue after waiting but being unable to get the service (imagine a client having a finite amount of patience).
4. A queue itself may throw out clients that have waited too long.
5. A queue may time out.

2.2.2.2 Reports Generated

A simulation is run in order to gain some insight into the behavior patterns of a system, so there must be some well-planned output. **ESSPRIT** provides for a range of screen dials, charts, graphs and reports as well as the capability to generate archival reports of the activities of all objects in the system. This capability is being included in the radar simulator.

2.2.2.3 Random Numbers

Pseudo-random number generators that are tunable and trustworthy need to be used. Unfortunately, these are not easily available off-the-shelf. Not only do horror stories abound (one random number generator was found that only output zero, for instance) but there are constantly articles in the

professional literature about how bad they are and how to do it right. The **ESSPRIT** simulator provides the capability to easily incorporate a suite of random number generators and to add a new generator whenever another may be required.

2.2.3 Scaling Up: Time Management

When small systems are scaled up to larger ones, there are bound to be surprises; nonlinearities abound. The problem that often hits simulation systems badly is the management of future event queues. These queues must support an *ordered set* so that whenever an item is removed from the set it is the one with the smallest time value. The obvious way to do this is to keep the list sorted and use an internal form such as a linked list or an array. The cost to enter an item into such a list whose size is M items is proportional to M—it is of time complexity $O(M)$. The cost to remove an item is independent of M—it is of order $O(1)$. Alternatively, one could keep the internal data structure unorganized, but then the cost to enter would be $O(1)$ and the cost to remove would be $O(M)$. The data structure that we suggest here is that used in the algorithm "heap sort." It is an array that is sorted only according to a very sparse constraint rule; namely, instead of requiring that $A_i > A_{i-1}$ for all i such that the expressions are valid subscripts, we require only that $A_i > A_{i/2}$. The heap sort idea, with its fewer constraints, is much easier to maintain. If M denotes the set's current population, then the cost to add a new element to the set is only $O(\log_2 M)$, and the cost to remove an item is also $O(\log_2 M)$. This is particularly striking in its effect: $\log_2 1,000)$ is approximately 10 and $\log_2 1,000,000$ is approximately 20.

2.2.4 Scaling Up Using Multiple Processors

As computers are reaching their ultimate limits as imposed by the speed of light and quantum mechanics, the obvious approach is to employ systems designed to do many things at the same time. Simplistically, N computers should be able to solve a problem in $1/N$ of the time a single computer would take. This is simplistic because N systems must generally cooperate, and the cost of coordination and communication can be prohibitive. As we shall see below, the problems involved in distributing discrete event simulation seem to as challenging as any in current computer science research. Several good

ideas are currently under investigation, but the general problem is far from solved.

There are two simple, quick and dirty suggestions to scale simulation systems up using multiple processors. The first is to use the observation that perhaps 80% of the time spent in typical large simulations involves what we have called the "miscellaneous" issues of simulation: time management, statistics gathering and reporting, and pseudo random number generation. These system components have very clear interactions with all the other components of a simulation, and separating them to other processors can easily be accomplished. The issues involved are the same as any other easy decomposition of a problem into parallel components.

The second easy approach is to design a systems simulation as a continuous event system. In this way, a global clock can broadcast the current simulated time to all components, the components can perform an increment of work, and parallelism can be achieved.

To distribute the objects of a simulated system over several processors, the current research suggests that the system needs to lose its global clock, and that every component (or processor) maintain its own local clock. The clocks can then be advanced, each independently of the others. The problem that must be avoided is for one component to change another's past. This is interpreted, in the local-clocks approach, as prohibiting a message of the form (m, t), where t is the time stamp, directed to component C in case component C has already issued a message of the form $(m', t'$ where $t' > t$.

Several methods have been suggested by researchers to avoid the problems associated with local clock coordination: roll-back, massive time messages, and (possibly) restricted time messages.

In case the system reaches a state where the violation of the time ordering of messages must occur, as when a component does receive a message "in its past," a message whose time stamp has a lower time than one that had been sent out by the receiver, then the receiver must initiate work to modify the advances made by the system in the time after the time stamp on the offending message. The simplest way to achieve this roll-back is via a checkpoint-restart procedure (familiar from the attempts to keep a computation going well beyond the mean-time-to-failure for a system.) This entails writing the entire state of the system to sa disk file periodically with the expectation that the system may have to revert to that state and make alternative decisions. An alternative to this sort of overhead is a more selective roll-back

wherein the component, whose past was detected to have been changed by the message with the low time stamp, attempts to revoke messages that are still in the unsent queue (like catching the mailman before he delivers something you wish you had not sent) and sending retraction messages to those recipients that had received the messages that it had tried to cancel. Such a effort could obviously get out of hand, with an ever increasing circle of messages and processes trying to back out of a contradictory situation.

Alternatively, a system monitor could try to detect the occurrence of such problems and keep them from occurring. The research in this direction seems to indicate an equally unacceptable amount of system overhead. One still does not gain substantial advantage from the parallelism.

The final method suggested is for system components to broadcast time coordination messages to the other components regarding its intention to update its local clock. This method also indicates a huge amount of message traffic, but it has suggested in its wake the following method: whenever a component wants to update its clock, it sends messages to the components that can send messages directly to it, essentially requesting permission to update the time. The requesting component elicits promises from those who send it messages that they will not send a message time stamped before a certain time. In order to answer such a request, these other components need to propagate the request backwards through the system to the components that might send messages to the .. Although this leads to interesting problems dealing with loops and parallel paths in the message graph, there seems to be a glimmer of hope that the overhead might not cancel out the multiple-processor gains.

Chapter 3

Model of the Radar Environment

A model of the radar environment was developed in the previous report [**Radar-89**]. An overview of the environment and the philosophy of the system will be reviewed here.

A radar environment can be simulated by providing models of four entities:

- Radar Module
- Environmental Module
- Target Module
- Environment

The modules can be described in object-oriented software, which permits as many radar transmitters, receivers and targets as desired to interact within the environment. The environment, itself an object, makes its presence felt through its modification of the signals used by the radar system to detect, track, classify and identify the targets.

The environment influences signals through *statistical* and *geometric* affects. The statistical effects include the multipath scatter off objects as well as the addition of random noise. The geometrical effects include attenuation through distance as well as scatter off particular point targets. All multipath effects could be handled geometrically, but the computational overhead

would slow the simulation. A statistical multipath effect combined with the effects of a few isolated point scatters will provide the same effects with a much smaller impact on simulation speed. Other geometric effects include distributed obstructions, such as clouds, which may provide an increased attenuation, and large reflecting surfaces. We have provided a mechanism to model the effects of clouds but have not modeled large reflectors.

In this system *targets* are aircraft or other flying objects which are to be detected, tracked, classified and identified by the radar system. The target models interact with the system by modifying the signals which arrive at their locations. The signals themselves are objects in the simulation, which permits them to be described in a variety of terms. The usual description includes their geometric origin (transmitter), amplitude, envelope function of time, carrier frequency and the phase function of time. These can be translated into other descriptions, such as in-phase and quadrature components. When a signal arrives at a target it is modified in a manner that is characteristic of the target and transmitted as a new signal. This signal eventually arrives at one or more receivers, where it can be processed and combined with other information for detection, tracking, etc.

The targets can also have geometric behavior which controls their flight paths. Speed, maneuverability, tactics and coordination with other targets can be modeled by defining the characteristics of the targets in the object-oriented simulation framework. In the long run, we expect that a large variety of target modules will be developed and stored in the system target library so that many scenarios can be modeled in an interactive simulation.

The radar module is made of transmitter, receiver and display modules. The transmitter and receiver modules are involved in the signal simulation process, but the display module is a passive module that is used only for the visual display of the simulated signal.

3.0.5 Transmitter and Receiver

Fig. 3.1 shows the overall structure of the radar module. As can be seen from the figure the transmitter and receiver modules receive input from two different sources, viz. Radar specifications module, and the Environment. The input specifications module provides system parameters such as frequency, peak and average power, polarization of the transmitted signal, coordinates, orientation and band-width of receiving and transmitting antennas, receiving

antenna temperature, and the synchronous time to synchronize the receiver and the transmitter. The input from the environmental module is the signals back-scattered from targets and various environmental clutter sources, along with motion induced Doppler shifts.

The receiver and transmitter modules are isolated. Fig. 3.2, shows a schematic diagram of the radar module with separate receiver and transmitter modules. The transmitter channels all its input information to the environment, in the form of signal objects. The internal structure of a signal object is described later in this report.

The display module simulates the visual feel of signal modifications caused by various specified environmental and target conditions. The display is very flexible and can show a wide range of parameters in geometric, statistical and other graphical formats.

3.1 Signal Objects

Signals are the active objects used to sense the presence of targets in a radar environment. There are so many possible signal designs and patterns of signals that it is impossible to provide a complete catalog. However, it is possible to approach the problem of signal design by using signal modeling tool that has sufficient flexibility to allow almost any signal that could be of interest to be described.

A general formulation of the signal waveform can be given as

$$s(t) = A_0 E(t - t_0) \cos[2\pi f_c(t - t_0) + \phi(t - t_0))]$$

The parameters are items in the description of the signal object:

- Carrier frequency f_c
- Signal Amplitude A_0
- Envelope function of time, $E(t)$
- Phase function of time, $\phi(t)$
- Time of origination, t_0
- Place of origination

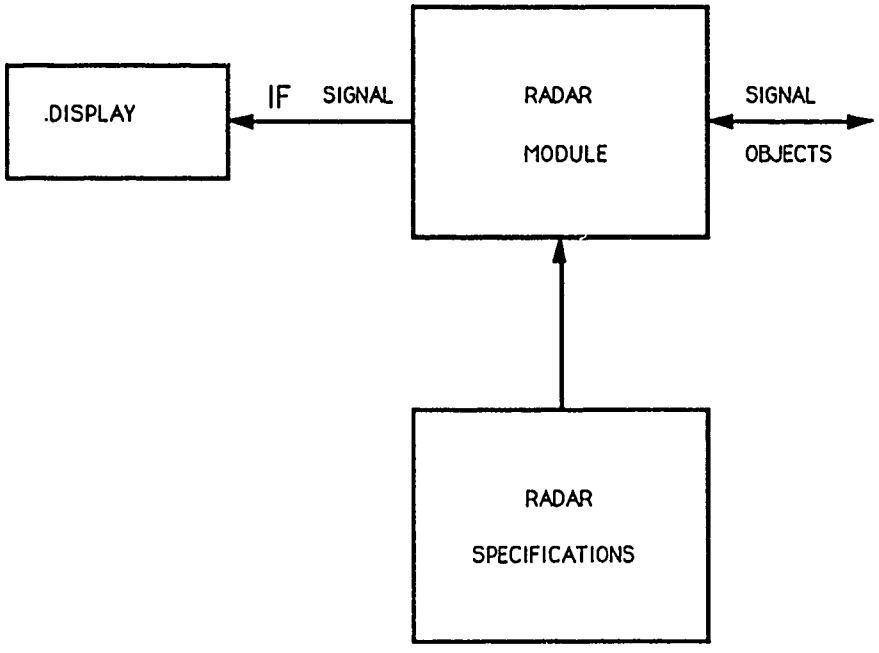


Figure 3.1: The radar module

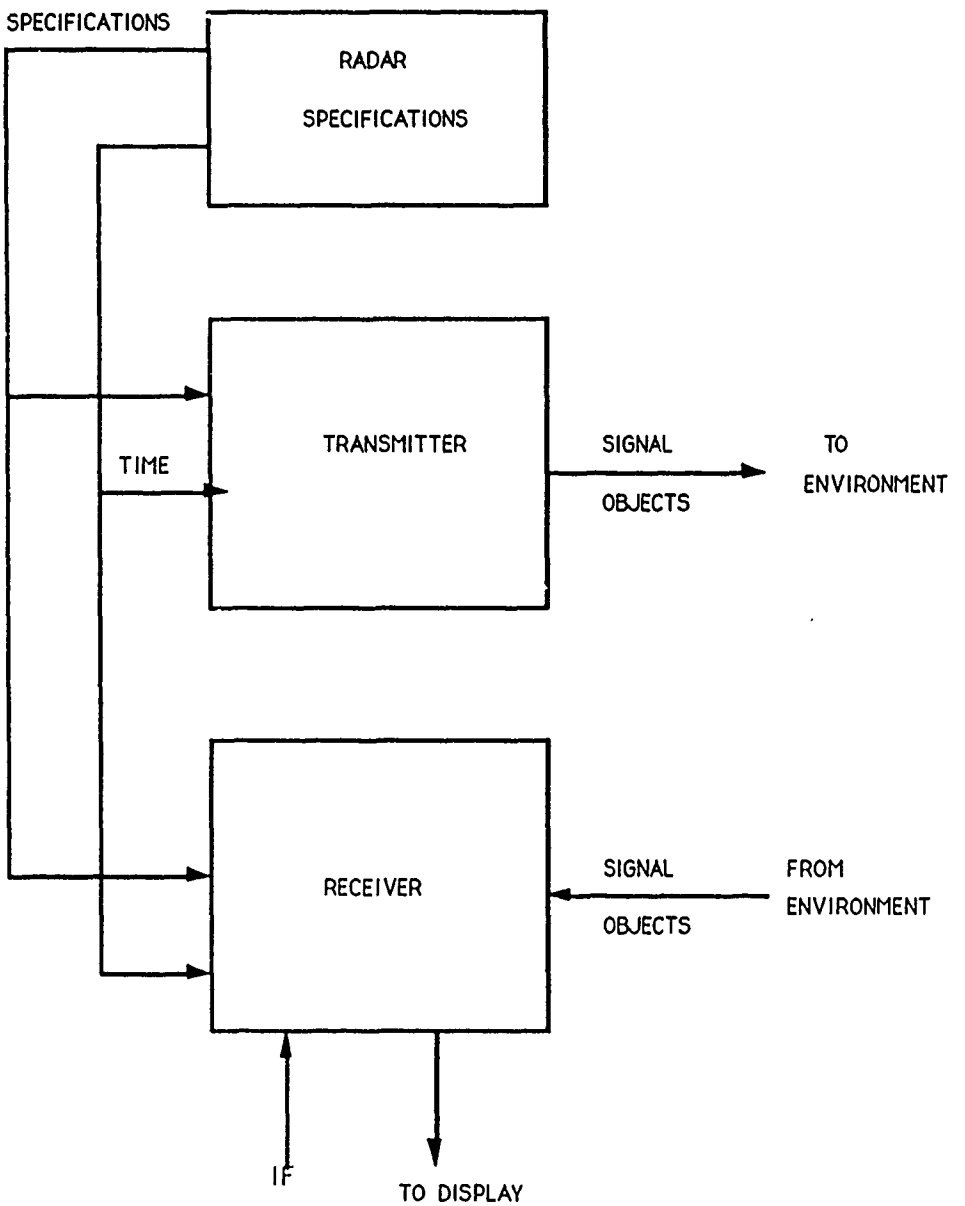


Figure 3.2: Schematic diagram of the radar module with separate receiver and transmitter modules.

The signal object may also carry other information, such as the identity of the transmitter and targets with which it has interacted.

The current implementation permits signal objects to be created by building new descriptions through the process of inheritance and specialization. The system has been tested using a basic set of pulse signal objects and the process of extending them to other more complicated signals has been illustrated by examples within the ESSPRIT simulation system.

Chapter 4

Status

The previous phases of the contract accomplished the following tasks:

1. Describe the multi-sensor surveillance environment in a manner which facilitates the transfer of techniques developed in other applications to its domain.

2. Identify the intelligent signal processing techniques which may be applied to the tasks of identification, sensor fusion for multiple targets and multiple sensors and intelligent tracking.

3. Identify concepts and techniques from other areas of artificial intelligence that will be required to provide the desired system performance.

4. Provide a road map for the development of system elements and a plan for integrating them into a functional body.

The current effort has extended this work to provide a model of the signal objects. Additional work is required in several additional areas. This can be carried out in a modular fashion and used to produce object descriptions or prototypes within the simulator environment. In this way systems can be compared and improved by competitive refinement and theory can be evaluated in an experimental domain. Some particular investigations include:

1. Develop a general model for signal objects.

2. Develop models for selected types of sensors.

3. Develop models for clutter and noise.

4. Develop models for jammers and ECM devices.

5. Develop models for knowledge structures which support sensor fusion by implementing the detection/correlation processor function.

6. Develop methods to maintain target track knowledge in a multi-sensor environment with multiple targets.

7. Investigate the effects of finite bandwidth between sensors in multi-sensor fusion systems.

8. Investigate knowledge structures for sensor fusion in a noisy, uncertain environment for target identification or target tracking.

9. Investigate methods to scale up parallel processing systems to achieve maximum performance.

10. Investigate methods to combine non-sensor knowledge or information, such as knowledge of tactics or intelligence reports of enemy activity.

11. Investigate methods to provide real-time situation assessment to field officers in a tactical environment.

The above list is only partial, but it illustrates the range of investigations that can be supported by *combining* AI techniques with modern simulation and prototyping tools. The combination will permit the results of diverse investigations to be built into a framework which can be used to carry out experiments and further investigations. The approach provides an open-ended tool for system modeling, analysis, and design.

4.1 Unix Implementation

The current system is constructed on a Texas Instruments Explorer Lisp computing framework. This provides an attractive development environment. However, the general lack of availability of this computer equipment makes it less than desirable for use in many research laboratories. We have recently developed a plan which would allow the system to be ported to a

Unix environment by using C++ and X-Windows. This system would have simulation features equivalent to those of ESSPRIT, but in a more common framework.

The porting of ESSPRIT to the Unix environment will enable the simulation system to be operated in a common signls lab environment. This project would be a logical step in the development of the simulation capability.

Bibliography

[Hendry-89] Hendry, Barbara. "Distributed Object-oriented Discrete Event Simulation," Master of Science Thesis Proposal, School of Computer Science and Technology, Rochester Institute of Technology, Rochester, NY. April 28, 1989.

[Radar-89] Rhody, H.E., Sher, D. and Modestino, J. "Intelligent Signal Processing Techniques for Multi-sensor Surveillance Systems," Final Report RADC Contract F30602-88-D-0027, Subcontract 3531245, Rome, NY. June 2, 1989.

Appendix A

The ESSPRIT Simulation Environment

The **ESSPRIT** (Explorer Simulation and Signal Processing system from the Rochester Institute of Technology) system developed at RIT Research Corporation is a software package written for the Texas Instruments Explorer Lisp Machine. **ESSPRIT** combines an object-oriented approach to simulation with a visual programming interface to provide high-level design capabilities along with the capacity for intelligent exploration of complex systems. Models which contain a large number and variety of interacting entities can be rapidly prototyped using this tool. **ESSPRIT** is presently being used at RIT Research Corporation to develop simulations in manufacturing, system dynamics, and radar design.

A.1 Visual Simulation

Traditional simulation languages require programmers to write large amounts of procedural code to model systems. Typically, an analyst using one of these languages develops a conceptual model of a system, which is passed on to a programmer who writes a program to implement the model. The analyst then receives the results of the simulation execution in the form of reams of statistics. The **ESSPRIT** system allows an analyst who is not an expert in simulation to interactively design, execute and analyze models of complex systems using a graphical interface. The topology of the objects and

transactions involved in a simulation is described using graphical tools. During execution, transactions are animated and selected statistics are displayed in graphs and gauges. At any time the simulation can be interrupted and altered. The result is a highly interactive system which allows an analyst to consider alternative hypotheses, modify models, and visually observe the dynamic behavior of a simulation.

A.2 Object-Oriented Framework

ESSPRIT is implemented using an object-oriented programming system. All data, programs, commands, and even the user are viewed as objects. Each object is an instance of a class, and each class has an associated collection of operations which can be applied to the object. The classes are arranged in a hierarchy so that subclasses can inherit properties of their parents. This approach provides for a high degree of code reusability and encourages the use of data and control abstraction.

A.3 System Architecture

ESSPRIT is composed of three major components: a graphical configuration editor, for designing simulations; a simulation executive, for running simulations; and libraries of simulation objects.

A.3.1 Configuration Editor

An **ESSPRIT** simulation is specified by drawing a block diagram of the system. Icons which represent instances of simulation objects are placed on the screen. An object can be constrained to communicate with another object by drawing an arrow to indicate a message path, or it may communicate with other objects in the simulation in a more generic manner. Each object contains parameters which may be adjusted before or during execution of the simulation. A variety of displays and gauges which monitor the state of the system or individual objects can be selected interactively. Any **ESSPRIT** diagram may be composed into a single object, which can then be used within another diagram. This allows a complex simulation to be con-

structed in a modular, hierarchical fashion, and provides the user with a way of investigating the behavior of a system at different levels of complexity.

A.3.2 Simulation Executive

ESSPRIT uses a process interaction approach to discrete-event simulation. Each object has a process which describes the sequence of operations through which the object passes during its life within the system. An object can be delayed either unconditionally for a certain period of time, or conditionally until a certain condition exists. The actual computation in an object's process may range in complexity from a simple Lisp expression to a large database query or expert system. The simulation executive allows objects to be run either sequentially or concurrently, and controls any animation and displays which are selected.

A.3.3 Simulation Object Libraries

The **ESSPRIT** configuration editor and simulation executive together form a "shell" which contains generic knowledge about simulation, but no specific knowledge about any applications. This domain-specific knowledge is contained in separate application libraries of objects and rules. In this way, a non-expert can build models composed of pre-defined parts while more sophisticated users can create their own objects or modify existing objects.

A.4 Hardware

Currently, **ESSPRIT** simulations run on a TI Explorer Lisp machine, which is a dedicated single-user system intended primarily for use in developing artificial intelligence applications. These computers are typically configured with 8-32MB of memory and a pair of 140 MB disk drives, and cost around $50K. The **ESSPRIT** executive is in the process of being modified to run simulations on a network of Explorers. Each object in a simulation will then be able to exist on any machine on the network. In addition, display and control interfaces to simulation objects may be spread over multiple machines. The **ESSPRIT** system is also being ported to a Macintosh II using Allegro Common Lisp.

Appendix B

ESSPRIT Technical Reference

The ESSPRIT Technical Reference was prepared as a separate document. Is included as a part of this report to provide technical information about the simulation prototype.

The ESSPRIT Technical Reference is also available as a separate document.

Appendix B. ESSPRIT Technical Reference

B.1 Introduction

This appendix provides details on the implementation of the simulation system developed for this project. First, the enhancements made to the Explorer window system are described. Second, the general simulation system is covered. Finally, the specific objects used in the radar simulation are described. Each section contains a description of the principal components involved, followed by a listing of the flavors, functions and methods which need to be understood by a programmer who wishes to develop new simulations or new applications which are similar to ESSPRIT.

B.2 The PX Window System

The standard windowing package supplied on the TI Explorer is designed for frame-based, non-overlapping windows. This approach is fine for an application with a fixed number of displays, such as an editor, but is inadequate for a more dynamic, graphically based application such as ESSPRIT. Thus, a number of additions were made to the Explorer window system to support the overlapping, resizable windows needed for the ESSPRIT system.

B.2.1 PX-Frame

The PX window system is activated by creating and selecting an instance of the flavor **px-frame.** If the PX system is loaded, this can be accomplished by entering SYSTEM-Z. Figure A1 shows a typical PX screen. A px-frame can contain any number of windows. All the windows which are visible at any given time belong to a particular **application.** An application is an instance of a system, such as an ESSPRIT simulation, which contains a collection of related objects. In this way a large number of windows can be managed with a minimum of clutter. In addition, multiple instances of an application can be executed simultaneously.

B.2.2 PX Applications

A PX application is a window-based interface to a particular collection of Lisp flavors, functions and objects. To utilize the PX window system with a particular system (such as ESSPRIT), the **px:application** flavor must be used. Once installed, a system which inherits from px-application will appear in the top half of the Application menu of a px-frame, and all instances of it will appear on the bottom half of the Application menu. Commands for a px-application can be defined using either the standard

Explorer ucl interface, or through px-menus (B.2.4). All windows used by a px-application should inherit the basic-px-window flavor to allow them to be accessed within the px-world.

B.2.3 PX Windows

Windows in the PX system can be moved, resized, closed, and zoomed. When a window is selected (by clicking anywhere in the window), highlight bars will appear in the window label. Only one window is the selected window at any given time. The selected window may be moved by clicking in the label, holding the mouse down, and dragging it to a new location. Clicking on the close icon in the upper left corner of the window will cause the window to be removed. Similarly, clicking on the zoom icon in the upper right corner will toggle the size of the window between its default size and its expanded size. The window can be resized arbitrarily by dragging the lower right corner of the window, which contains an invisible grow box. This will expand or shrink the window to the desired size.

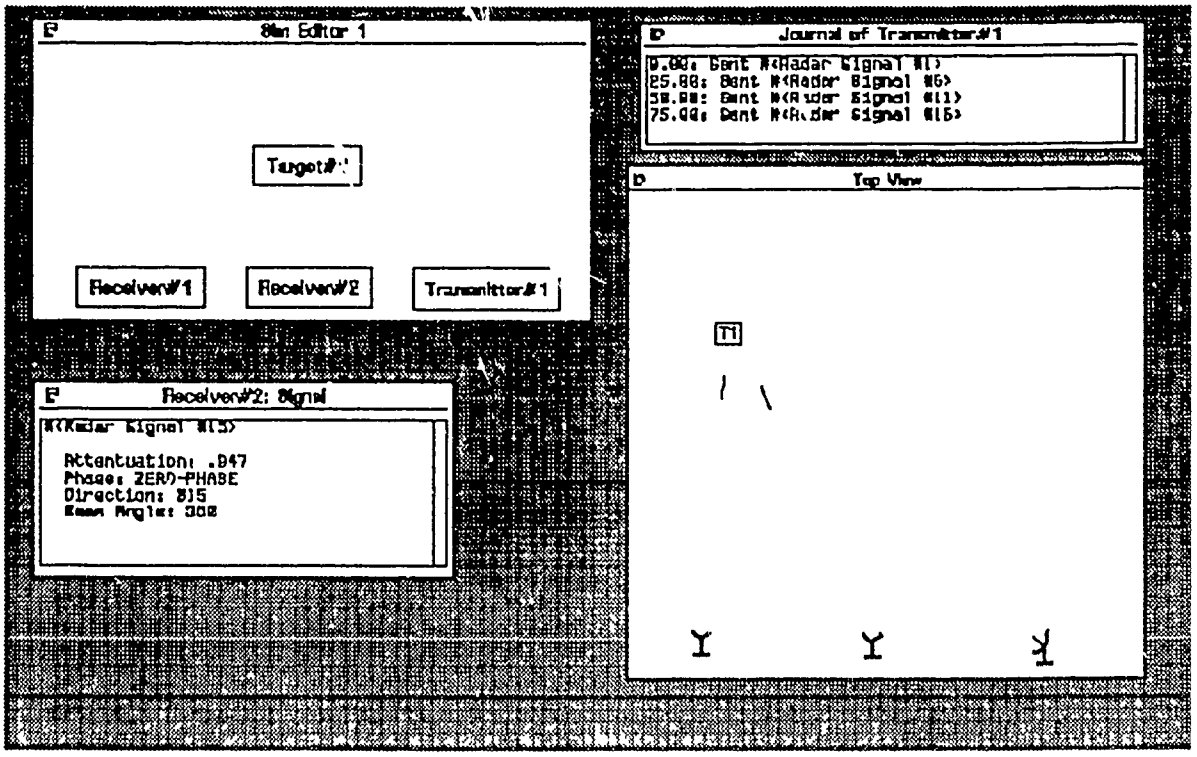


Figure A1. Example PX/ESSPRIT Screen

A number of basic types of windows are provided in the PX system for use in applications. The basic-px-window flavor provides the essential

hooks to a PX application to enable window selection, window movement and command enabling. Each PX window may modify an application's menus when it is selected. Typically, a window will enable the commands which apply to the window, and disable those which do not. The **button-mixin, px-label-mixin, close-box-mixin, zoom-box-mixin, grow-box-mixin** and **collapse-box-mixin** flavors provide window manipulation capabilities in any desired combination. A large number of graphical objects and functions are available with the **px-graphics-window** flavor. The standard graphical objects include lines, rectangles, circles, arcs, text, and so on. Standard functions include object insertion, deletion and selection; rubberbanding, dragging and resizing; and rasterizing. For dealing with regular text, the **px-scroll-window** flavor provides a scrolling window which displays a list of text items.

B.2.4 The PX Menu Bar

Commands for a px-application can be organized into a collection of menus, which will appear on the PX menu bar. Clicking on the title of a menu in the menu bar will pop up the menu. Although the Explorer is equipped with a three button mouse, in the PX system all mouse clicks are interpreted as a left click. A menu item can be chosen by sliding the cursor down the menu and clicking again on the desired item. Keystrokes can also be assigned to menu commands by assigning the ucl keystroke as an item's action in either the function **define-menu** or the function **make-menu-item**.

B.2.5 Flavors, Functions and Methods

This section outlines the principle flavors, functions and methods used in the PX window system.

B.2.5.1 Basic PX Flavors and Functions

px:px-frame **[flavor]**

This is the top-level window created by the PX window system. An instance of px-frame can be created using the system key SYSTEM-Z, through the System Menu, or by the function **px:make-px-frame**.

px:make-px-frame **[function]**

Creates and selects an instance of px:px-frame.

px:application **[flavor]**

Each package designed to work within the PX window system must include a flavor which inherits from px:application. This flavor should provide the principle application menus as the default initialization.

px:find-application application-name **[function]**

Locates an existing application with the given name (a string). Normally, applications are given enumerated names in order of their creation, such as "Simulation#1". Note: when using remote simulations, the full simulation name, such as "Simulation#1@clark", must be parsed before using find-application.

B.2.5.2 PX Window Flavors and Methods

basic-px-window **[flavor]**

Provides essential window selection, menu enabling and disabling, and mouse handling.

px-label-mixin **[flavor]**

Draws a centered label at the top of a window, surrounded by selection bars when the window is the selected window.

close-box-mixin **[flavor]**

Places a close box (in the form of a small RIT logo) near the left edge of the label area. A mouse click on this box will cause the window to be removed.

zoom-box-mixin **[flavor]**

Puts a small circle near the right edge of the label area. A mouse click on this circle will expand the window to the full screen, or contract it down to its original size and location.

grow-box-mixin **[flavor]**

Provides window resizability. The box is invisible, but occupies a small square region in the lower right corner of the window. When the mouse is held down over this box and then moved, the window boundary changes size to follow the mouse. When the mouse is released, the window is redrawn to the new size.

collapse-box-mixin **[flavor]**

Puts a small zoom icon at the right edge of the label area. When the mouse is clicked over this icon, the window will collapse down to its label. The collapsed window then can be moved off to the side where it

won't interfere with other windows. When the collapse box is cl.. ed again, the window will expan. to its original size and position.

px-graphics-window [flavor]

Provides a variety of graphical objects and operations.

:cut [method of px-graphics-window]

Removes the currently selected objects and places them on the clipboard.

:copy [method of px-graphics-window]

Places a copy of the selected objects onto the clipboard.

:paste [method of px-graphics-window]

Inserts the objects on the clipboard into the window.

:selected-objects [method of px-graphics-window]

Returns a list of the objects which are currently selected in the window.

standard-px-window [flavor]

This flavor combines px-graphics-window with close-box-mixin, label-mixin, grow-box-mixin and collapse-box-mixin to provide a standard window with graphical capabilities.

px-scroll-window [flavor]

Provides a text pane which displays a scrolling list of text items.

:add-item item-text [method of px-scroll-window]

Appends a string to the end of the list, and updates the window to make this item visible.

:set-items items [method of px-scroll-window]

Sets the window item-list to items, which should be a list of strings, and updates the window to display the head of the list.

:clear [method of px-scroll-window]

Removes all items in the list, and refreshes the window.

B.2.5.2 Menu Flavors and Functions

command-menu **[flavor]**

Basic flavor for menus which pop up from the menu bar. New instances are created with the define-menu function.

menu-item **[flavor]**

Flavor for elements of a menu; created with the make-menu-item function.

define-menu menu-name &key title items **[function]**

menu-name	a symbol
title	a string, which will appear in the menu bar
items	a list of menu item descriptors which specify the string to appear in the menu and the corresponding action

make-menu-item name &key value font label keystroke segment-number **[function]**

name	a symbol
value	the value to be returned
font	the font in which the label is displayed
label	a string
keystroke	keystroke equivalent
segment-number	which part of the menu the item should appear in; segments are separated by lines

remove-menu-item menu-name item-name **[function]**

Removes the item named item-name from the menu named menu-name.

add-menu-item menu-name item **[function]**

Adds an item to the menu named menu-name.

enable-menu-item item-name **[function]**
disable-menu-item item-name **[function]**

Enables or disables the item with the given name.

B.3 The ESSPRIT System

The simulation system developed for this project, ESSPRIT, is a general purpose discrete event object-oriented simulation system written in Lisp with Flavors.

B.3.1 Introduction

A ESSPRIT simulation consists of a collection of objects which interact over time by sending messages and creating new objects. The simulation is specified by graphically constructing the objects and illustrating their relationships via arrows. A simple simulation is shown in Figure A2 below.

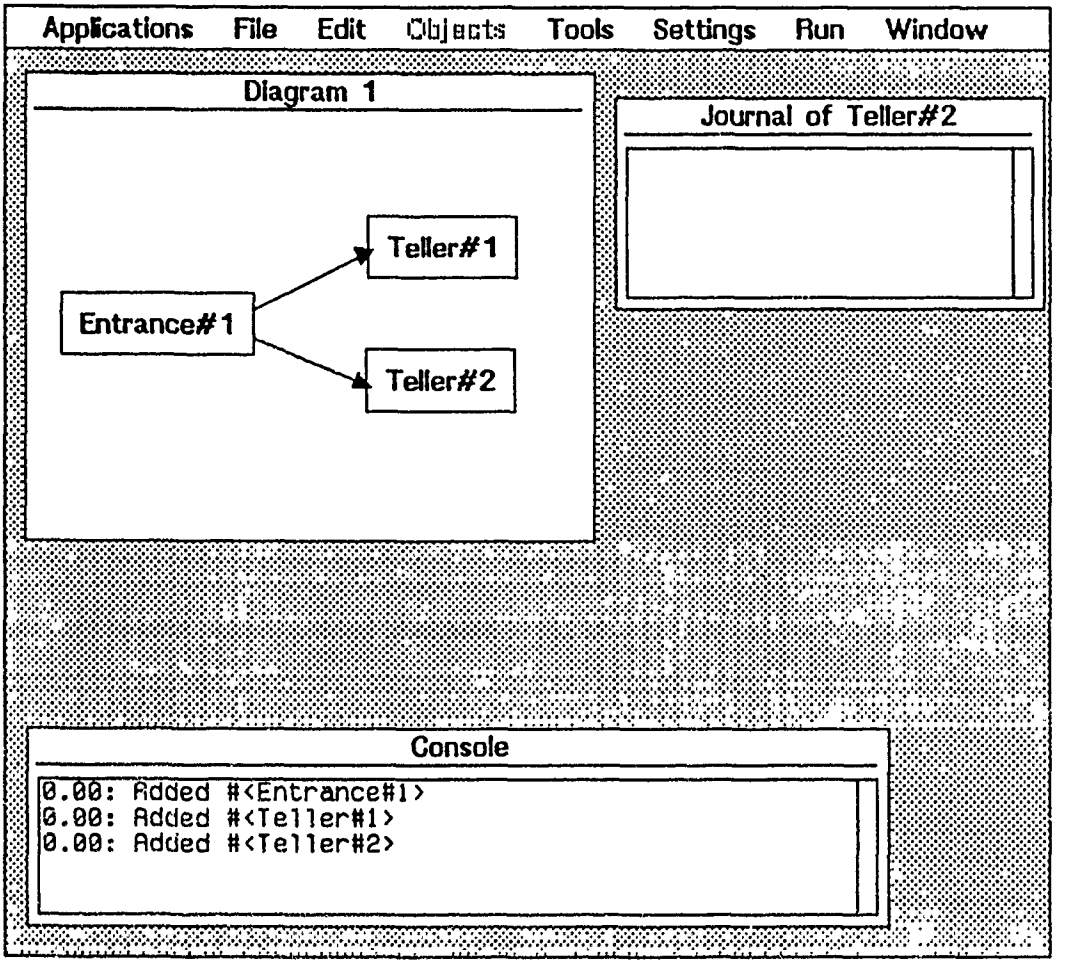


Figure A2 Simple ESSPRIT Simulation

B.3.2 ESSPRIT Objects

ESSPRIT objects may be permanent or temporary. In addition, an object can create other objects, and an object can die. The actions which an object performs over its lifespan are described in a method (**:run**) which is repeated until the simulation is terminated or the object dies. Each object runs in its own process. On a sequential machine, such as an Explorer, this means that the objects are sharing a single processor. However, this architecture makes it easier to take advantage of multiple processors, either on a single machine or over a network. Each object contains a message queue, into which come messages from other objects.

The flavor **sim:basic-object** provides all of the essential operations which an object needs to perform in a simulation, including message-passing, queue management, parameter editing, process management, history and statistics collection, reporting, and image displays. In addition, common types of objects such as sources, sinks, servers and routers are provided. Thus, new classes of objects need only to describe the parts of their behavior which are unique.

Each object has a unique name, determined by its flavor name and the time at which it was created. For example, Teller#2 would be the second teller object created in a simulation. In a networked simulation, the extended **contact-name** also contains the host machine name and simulation name. For example, Teller#2.Simulation#3@clark would be the second teller created in Simulation#3 on the machine named clark.

B.3.3 The ESSPRIT Executive

Each simulation is controlled by a simulation executive, which is an instance of the flavor **sim:simulation**. A simulation is also a PX application, which means that it handles the command interface for the windows it contains (see B.3.12 for a listing of the command menus). The executive is responsible for managing the clock, locating objects, inserting and deleting objects, handling errors, and gathering statistics. In addition, in a networked simulation the executive directs all messages between server objects and client objects.

The ESSPRIT executive is process-oriented. To begin, the executive notifies each active object that the simulation has started. It waits until all of the objects have stopped, and then determines the next clock time by examining each object's wake-up time. It advances the clock to the lowest time, and repeats the process until a maximum time is reached or the simulation is interrupted with a pause or abort.

Each object in a simulation executives its **:run** method in a separate process. Typically, an object will perform some action and then wait for either a specified length of time, until some time, until some condition

becomes true, or until it receives a message. This synchronization is provided by the set of wait macros, which include **wait-for-time, wait-until-time, wait-for-message, wait-for-any-message** and **wait-for-condition.**

B.3.4 Creating a Simulation

A simulation in the ESSPRIT system is constructed using a set of windows which collectively described the initial objects together with their relationships. Typically, a diagram-editor is opened to begin defining the simulation. The domain and object library are then specified through the Settings Menu. The objects which are desired can then be chosen off of the Object Menu. When an object is chosen, an icon representing an instance of the specified class appears in the diagram-editor. This icon can then be moved around within the window.

To modify the parameters of an object, just double-click on its icon. A standard Lisp edit-parameters dialog will then appear. Some of the parameters may be of simple type, such as strings or numbers. These can be edited by keyboard input. Other parameters, such as those which require a random distribution, may be constrained to a menu selection.

To indicate a message path from one object to another, first pick the path tool. The cursor will change to a pencil to indicate the mode change. Now click on the sender object, and drag the path line to the receiver. The exact meaning of this path may vary depending on the types of objects involved, but in general it means that the receiver will get any messages generated by the sender.

B.3.5 Running a Simulation

A simulation is started via the menu command **Start.** While executing, each object may output messages to its journal or to the console, and animate itself using one or more of its images in a graphics window. A simulation can be temporarily interrupted with the **Pause** command. At this point, objects can be inspected or modified, or new objects can even be added. After pausing, the simulation can be started up again where it left off via the **Continue** command. At any point the **Stop** command can be used to reset the simulation.

B.3.6 Object Communications and Queues

ESSPRIT objects communicate by sending messages using the **send-message** macro. When a message is sent to an object, the message is put into the object's message queue (an instance of the **message-queue** flavor). It is the responsibility of the recipient to periodically check its message queue and handle any messages which have entered. A queue management strategy,

such as LIFO or FIFO, can be specified for the message queue. When an object wishes to examine the next message in its queue, the **:next-message** method should be used to insure that the proper strategy is always used.

In a simulation in which all the objects exist on a single machine, messages are passed directly from the sender to the receiver. In a networked simulation, messages between two objects which are on the same machine are handling directly. If the receiver of a message is on a different machine, the message passes through the objects' simulations in intermediate steps.

B.3.7 Standard ESSPRIT Windows

A number of different types of windows are provided in ESSPRIT for creating, executing and analyzing simulations. A **diagram-editor** is a window in which object icons can be moved around, edited and connected with paths to other objects. The paths in a diagram-editor indicate how objects are related, and how messages (and dynamic objects) are to flow in the simulation. In a **map-editor**, the placement of icons determines the physical location of the objects. In a **plot-editor**, the placement of object icons determines the values of some common parameter or pair of parameters.

During execution, the **console** displays system-level output which describes primarily the operation of the executive. A **clock-window** can be displayed to show the progression of the clock during the simulation. Each object can display its actions textually in a **journal-window** or graphically in a **dial-window**. In a networked simulation, each client can be monitored with a **remote-console** window.

After a simulation has been completed or interrupted, statistics can be viewed textually in a **statistics-window**, or graphically in a **plot-window**.

B.3.8 Random Distributions

Random number generation is an important component of a simulation package. In ESSPRIT, any parameter which is to be drawn from a random distribution can be described through the edit-parameters dialog. In this dialog the distribution can be chosen from a menu, and the parameters to the distribution can be entered as well. The distributions which are provided in the ESSPRIT package include the following:

- uniform
- normal
- triangular
- exponential
- poisson
- erlang

- lognormal
- gamma
- beta
- weibull

B.3.9 Statistics

Each simulation object has the ability to maintain a variety of statistics about itself throughout the course of a simulation. An object's message queue can keep simple statistics concerning its length and waiting time. An object can also place entries into its history list. A particular variable within an object may be specified for detailed report. In this case, each time this variable changes an entry is added to the history list. When the simulation is done or interrupted, a statistics window or plot can be opened to display this accumulated information.

B.3.10 Networked Simulations

A host machine can be specified for each ESSPRIT object. The machine on which the simulation is created is referred to as the server, and other machines which are involved in the simulation are referred to as clients. The simulation on the server is called the main simulation, and those on clients are referred to as remote simulations. When a networked simulation is started, an instance of **sim:remote-simulation** is created on each client. The remote objects are then created on the remote simulations, and all of the simulations are started. Each remote simulation will run independently until its objects have stopped. It then waits for the main simulation to notify it of the next time. The main simulation waits until its own objects and all the remote simulations have stopped before advancing the clock.

The main simulation also serves as the conduit for all messages which pass between objects which are on different machines. A remote message first goes to the object's simulation, then on to the main simulation, then to the receiver's simulation, and finally to the receiver.

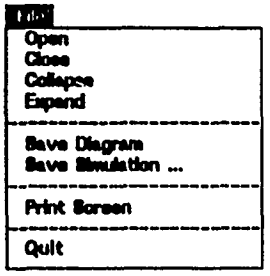
B.3.11 Object Libraries

Two facilities are provided to help organize the various objects which can be used in ESSPRIT. A **library** of objects can be defined using the **define-library** macro. The Object Menu contains the objects which are in the current library. A collection of libraries in turn can be grouped into a **domain**. The current domain can be set using the Settings Menu.

B.3.12 Menus

This section contains a summary of the menus which appear on the menu bar in the ESSPRIT system.

B.3.12.1 File Menu



Open - opens a diagram or simulation which has been saved to disk
Close - closes the selected window
Collapse - collapses the selected window to its label
Expand - expands a collapsed window to its original size and position
Save Diagram - saves a diagram to disk
Save Simulation - saves the current simulation to disk
Print Screen - sends a screen dump to the printer
Quit - exits from the PX system

B.3.12.2 Edit Menu

Cut - deletes the selected objects in the current window, and places them on the clipboard
Copy - places a copy of the selected objects on the clipboard
Paste - inserts the contents of the clipboard into the selected window

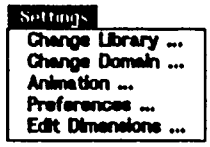
B.3.12.3 Tools Menu

Arrow - for selection
Pencil - for drawing paths
Text - for inserting text

B.3.12.4 Object Menu (variable)

This menu contains the objects which are contained in the current library. Selecting an item on this menu will insert an instance of the chosen class into the current window.

B.3.12.5 Settings Menu



Change Library - allows the current library to be selected from a menu of libraries in the current domain
Change Domain - allows the current domain to be selected from a menu of all domains
Animation - provides a menu of animation types for the current window
Preferences - allows assorted user preferences to be modified
Edit Dimensions - for a map window; allows x and y intervals to be modified

B.3.12.6 Run Menu

Start - begins the simulation
Stop - aborts the simulation (cannot be resumed)
Pause - interrupts the simulation
Continue - enabled after pausing; resumes the simulation
Backup (not yet implemented) - backs up the simulation to the specified time

B.3.12.7 Window Menu

This window contains commands to create new windows of several different types together with a list of the existing windows.

New Diagram
New Map
Console
Remote Console
Journal
Statistics

B.3.13 ESSPRIT Flavors, Functions and Methods

This sections contains a brief description of the principal flavors, methods and functions used in the ESSPRIT system.

B.3.13.1 Basic Flavors and Functions

sim:simulation [flavor]

Manages all the objects in a simulation, keeps track of the time and handles error conditions

sim:clock [flavor]

Maintains the current simulation time.

B.3.13.2 Object Flavors and Functions

sim:basic-object [flavor]

The fundamental flavor which should be inherited by every ESSPRIT object. It is composed of the following flavors.

sim:journal-mixin [flavor]

Provides the ability of an object to display text messages in a journal window.

:report format-string &rest args [method of journal-mixin]

This method is used to display a text message in a journal window. The formatted string will be added to the end of the list of items in the journal.

sim:slot-mixin [flavor]

Manages the parameters and variables of an object. Parameters are instance variables are constant for the duration of a simulation, while variables are instance variables whose values are expected to change. Both types can be modified when the edit-parameters dialog for an object is displayed.

sim:sim-process-mixin [flavor]

Allows an object to run in a separate process.

sim:image-mixin [flavor]

Provides the ability of an object to display itself in different graphical forms in different types of windows. For example, most objects will display themselves as a block-image in a diagram, and as small circles in a plot-diagram.

sim:message-queue-mixin [flavor]

Manages the message queue of an object according to a specified protocol, and maintains statistics about its activities.

sim:history-mixin [flavor]

Maintains a list of history items for analysis and generation of statistics.

sim:network-mixin [flavor]

Provides an extended naming mechanism for objects in a networked simulation.

B.3.13.3 Synchronization Macros

wait wait-list [macro]

The basic wait macro, which is used by all other wait macros. The wait-list is either a primitive wait-list or a list of primitive wait-lists. A primitive wait-list is a list of two items. The first item is a keyword which describes the type of waiting to perform; this can be either :time, :object or :message. The second item is an argument to the first.

wait-until-time wake-up-time [macro]

Causes the object to wait until the simulation clock time becomes wake-up-time.

wait-for-time elapsed-time [macro]

Causes the object to wait for a specified length of time.

wait-for-message message [macro]

Causes the object to wait until the specified message has been received in the message-queue.

wait-for-any-message [macro]

Causes the object to wait until any message has been received.

wait-for-condition condition [macro]

Causes the object to wait until the given condition becomes true.

B.3.13.4 Communication Flavors and Functions

sim:message-queue [flavor]

Contains a list of messages ordered according to a given management strategy, such as LIFO or FIFO.

sim:message [flavor]

Contains information about a message, including the sender, receiver, method and arguments. The sender and receiver can either be instances of objects or contact-paths.

sim:send-message recipient message [macro]

Creates an instance of sim:message and routes it to the recipient.

B.3.13.4 Networking Flavors and Functions

sim:remote-simulation [flavor]

The executive for the portion of a simulation which is run on a client machine. A remote-simulation manages its local objects just as the main simulation does, except that it gets the next clock time from the main simulation.

sim:remote-connection [flavor]

Contains the eval-streams which provide the low-level network connections between the main simulation and its clients.

sim:contact-name [flavor]

Contains the name of an object, the simulation it is running in, and the host machine.

sim:find-simulation simulation-name [function]

Locates the simulation with the specified name on the local machine.

B.3.13.5 Window Flavors and Functions

sim:basic-editor [flavor]
sim:diagram-editor [flavor]
sim:map-editor [flavor]
sim:plot-editor [flavor]

These windows are used to create a simulation. When an instance of one of these windows is selected and an object flavor is chosen from the Object Menu, an instance of the object flavor will be created and its image will appear in the window. This image can then be dragged to a position within the window. In a map-editor or plot-editor, this position corresponds to variables within the object. In a diagram-editor, paths can be drawn between objects to indicate message paths.

sim:journal [flavor]

A kind of px-scroll-window which is used by an object to display text messages about its activity while a simulation is running.

sim:console [flavor]

Similar to a journal, but used by the main simulation for reporting system messages.

sim:remote-console [flavor]

A journal for displaying messages from a remote simulation.

sim:statistics-window [flavor]

Displays a collection of statistics in text form for a given object after a simulation has been completed or interrupted.

sim:clock-window [flavor]

Displays the current simulation time.

sim:plot-window [flavor]

Displays a plot of a selected variable for an object or collection of objects over the course of a simulation. This can be created either during a simulation, in which case the plots are generated as the simulation progresses, or after the simulation has stopped.

B.3.13.6 Domain and Library Flavors and Functions

sim:domain [flavor]

Allows simulation object flavors to be grouped, and special windows to be defined. A domain contains a set of libraries and a list of window specifications.

sim:define-domain name &key parameters window-specs [macro]

name	a string
parameters	a list of common parameter or variable names which can be used in a plot

window-specs	a list of items which describe the special windows associated with a domain; each item is a list of the form (window-name window-flavor label x y width height)

sim:library **[flavor]**

Contains a collection of object flavors. The flavors in the current library will appear on the Object Menu.

sim:define-library name &key domain flavors documentation **[macro]**

name	a string
domain	the domain containing the library
flavors	a list of object flavors
documentation	a string

B.4 Radar Simulation in the ESSPRIT System

A prototype of a multi-sensor, multi-target radar simulation has been implemented in the ESSPRIT system. Any configuration of radar transmitters, receivers and targets can be created using ESSPRIT diagrams and maps. Special graphical output has been provided for animating the radar simulation and displaying the signals which are generated, modified and processed.

B.4.1 Creating a Radar Simulation

An ESSPRIT radar simulation is created like other ESSPRIT simulations, using a collection of windows to describe the basic objects involved. For radar, the objects available are transmitters, receivers and targets. The initial physical arrangement of the chosen objects can be described (in 2D form) in a map-editor, and journals and other displays can be selected. An example of a simple radar simulation is shown in Figure A2.

B.4.2 Radar Displays

A radar simulation can make use of the standard ESSPRIT simulation windows, including journals, consoles, plots, and so on. In addition, there are three windows which are specific to radar: **radar-top-view**, **radar-waveform-window**, and **radar-signal-window**.

The radar-top-view window provides a 2D view of the objects in the simulation. As signals are generated by transmitters and travel outward, they are animated as either rays or arcs (see Figure A2). Targets are also animated as they move, but at a much slower rate. A radar-waveform-window can be opened for a given transmitter or receiver. This window will display a waveform plot of the signals which are generated or received. A radar-signal-window is similar, except that it displays in textual form the parameters which define the signals.

B.4.3 Executing a Radar Simulation

A radar simulation is executed like other ESSPRIT simulations. After configuring the initial transmitters, receivers and targets, the simulation is initiated with the Start command. The simulation can be paused, and parameters of the receivers, transmitters or targets can be modified if desired. After a simulation has been completed, statistics from each of the objects can be examined and displayed.

B.4.4 Radar Objects

This section describes the principal flavors and methods used in the ESSPRIT radar simulation.

radar-signal **[flavor]**

Radar signals are the active objects used to sense the presence of targets in a radar environment. Signals are represented in a parametric fashion, and include the following information:

```
carrier-frequency
signal-amplitude
envelope-function
phase-function
time-of-origination
place-of-origination
```

A radar signal is generated by a transmitter according to the transmitter's attributes. It then determines

:find-hit **[method of radar-signal]**

Examines the radar environment and locates the closest object that the signal will hit.

:hit-object **[method of radar-signal]**

Moves the signal to the object, showing the movement with animation in the graphics window, and notifies the object of the impact.

:compute-samples **[methods of radar-signal]**

Generates a sequence of sample points from the parametric description for display in a radar waveform window.

radar-waveform-window **[flavor]**
radar-signal-window **[flavor]**

Used to display the radar signals which are generated by a transmitter or received by a receiver. A waveform displays a sampled version of the signal, while a signal window displays the parameters of the signal in text form.

radar-transmitter **[flavor]**

A transmitter is an originator of radar signals. A transmitter periodically emits a signal whose characteristics are determined by the transmitter's parameter settings. These parameters include frequency, peak and average power, polarization, orientation and bandwidth.

:send-signal **[method of radar-transmitter]**

Generates a radar signal with the specified properties, and sends it into the radar environment.

radar-receiver **[flavor]**

Contains a processing function which is applied to incoming signals. In addition, environmental influences such as multipath effects, clouds and attenuation can be incorporated in the receiver.

:receive-signal **[method of radar-receiver]**

Processes a signal according to the processing-function parameter. Note: in the future, the processing will be specified with a data-flow diagram as in the ESPRIT speech processing system.

target **[flavor]**

Targets are aircraft or other flying objects which are to be detected, tracked, classified and identified by the radar system. A target creates new signals when hit by radar signals. It contains a description of how it will interact with signals, which includes the amplitude, envelope function, carrier frequency and phase function. In addition, a target has geometric attributes which control its flight path.

:reflect-signal **[method of target]**

Creates a new signal based on the interaction of the target and the incoming signal. The baseline :reflect-signal method attenuates the signal by a specified amount, and adds noise. Flavors which inherit from target, such as flavors for particular aircraft or classes of aircraft, will modify this method to return a signal consistent with the target's radar signature.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.